CS 305
Design and Analysis of Algorithms

09 / 14 / 2021

Instructor: Michael Eckmann

Today's Topics

- Questions/Comments?
- Arithmetic series
- Selection sort analysis
- Insertion sort pseudocode and analysis
- MergeSort
- Recurrence Relations
- Logarithms

Arithmetic Series

- Arithmetic Series
- This is one of 3 sums you should memorize.
- The other two being Geometric and Harmonic.
- See Appendix A of our text (pgs. 1146, 1147)

Arithmetic Series

Let's prove the sum is big $O(n^2)$ on the board

I leave you to prove that it is also Big Omega of n^2

Together, they mean that it is Big Theta of n^2

Sorting algorithms

- Reminder of what Selection Sort and Insertion Sort do (online animation).
- Let's discuss Selection Sort (referred to on p. 29 2.2-2).
- Keep current "first".
- Continually find min element index in list and swap with the current "first".
  - How long does 1 find min take?
  - How many times do we do find min?
  - Are there any best or worst case situtations? All the same?

Sorting Algorithms

- Insertion Sort pseudocode on the board with run time analysis.
- Are there any best or worst case situtations? All the same?

Sorting Algorithms

- Let's look at an instance of Merge Sort and see how it works.
- Are there any best or worst case situtations? All the same?
- Let's see a recursion tree for n elements of MergeSort.
  - How many elements processed in each level?
  - How many levels?

Sorting Algorithms

- Additional comments on the sorts
  - InsertionSort and SelectionSort are both in-place algorithms and have small constants (as compared to MergeSort)
    - InsertionSort and SelectionSort each require extra space Big Theta (1)
  - MergeSort is NOT in-place (requires n additional space) and has large constants
    - MergeSort extra space analysis: Big Theta (n)

Recurrence Relations

- For MergeSort
- Time(n) = 2 * Time(n/2) + n
  - Number of recursive calls is 2
  - The time for one recursive call is Time(n/2)
  - The time in a call is n
- In general for divide and conquer
- T(n) = a * T(n/b) + f(n)
  - Number of recursive calls is a
  - n/b is size of list in a recursive call
  - The time in a call is f(n)

Recurrence Relations

- In other words, a divide and conquer algorithm that creates *a* subproblems each a factor of *1/b* the size of the original problem and takes *f(n)* amount of time to do the divide and combine steps.
- T(n) = a * T(n/b) + f(n)
  - Number of recursive calls is a
  - n/b is size of list in a recursive call
  - The time in a call is f(n)

Recurrence Relations

- Let me draw the recursion tree for arbitrary a and b.
- How many leaves?
- Let's see an example with explicit values for a and b.
- How many leaves?
- $T(n) = a * T(n/b) + f(n)$
- Let's prove, by induction, that the number of leaves in a perfect binary tree is 2^d, where d = depth of the tree.

Logarithms

$y = \log_b a$ iff $b^y = a$, $a > 0$, $b > 0$ and $b \mathrel{!}= 1$

- In English: one way to read $\log_b a$ is "what exponent of b results in a?"
- When b=2, $\log_2 a$ is the number of times a can be cut in half (until you hit 1.)
- $\log_2 1024 = ?$ It is the power of 2 that gives you 1024 (or the number of times 1024 be cut in half until you hit 1.)

Logarithms

- Let's show some properties of logarithms on the board.

Logarithms

- Let's go back and look at the a-ary tree and recall there are a^($\log_b n$) --- that is a to the power $\log_b n$
- Any other way to specify a^($\log_b n$) ?

Logarithms

- Let's go back and look at the a-ary tree and recall there are a^($\log_b$n) --- that is a to the power $\log_b$n
- Any other way to specify a^($\log_b$n) ?
- n^($\log_b$a)
- better because it is n to some constant power

Master Method for Recurrences

- Soon we will learn the Master Method which can be used to solve recurrences of the form: $T(n) = a * T(n/b) + f(n)$
- There are 3 cases to consider each based on a relationship of time spent at leaves vs. at the root.
- As an example, and to help us shortly, let's now count the number of nodes in a perfect tree, say a 3-ary tree.